(4)

The Programmer's Guide to Moviola: An Interactive Execution History Browser

Robert Fowler and Ivan Bella

Technical Report 269
February 1989

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

The Programmer's Guide to Moviola: An Interactive Execution History Browser

Robert Fowler
Ivan Bella

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 269

February 1989

Abstract

Moviola is an interactive browser used to create, examine, and manipulate graphical representations of synchronization histories of concurrent programs. It is part of an integrated, programmable toolkit for debugging and performance tuning parallel programs. This guide presents *Moviola* by describing its use as a standalone program and as a component of the toolkit. In addition, we describe the interface seen by a programmer of the toolkit.

This work is supported in part by U. S. Army Engineering Topographic Laboratories research contract DACA 76-85-C-0001, in part by ONR research co... ...s N00014-91-K-0655 and N00014-87-K-0548, and in part by NSF research grant CCR-8704492.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 269	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): The Programmer's Guide to Moviola: An Interactive Execution History Browser	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Robert Fowler and Ivan Bella	8. CONTRACT OR GRANT NUMBER(s): DACA76-85-C-0001, N000 14-87-K-0548, N000 14-84-K-0655
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Department, 734 Computer Studies Bldg, University of Rochester, Rochester, NY 14627	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: D. Adv. Res. Proj. Agency, 1400 Wilson Blvd., Arlington VA 22209	12. REPORT DATE: February 1989
	13. NUMBER OF PAGES: 28
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Res., Information Systems, Arlington, VA 22217; US Army ETL, Fort Belvoir VA 22060	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

visualization, performance analysis, multiprocessor, debugging, program replay, parallel programming environments

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Moviola is an interactive browser used to create, examine, and manipulate graphical representations of synchronization histories of concurrent programs. It is part of an integrated, programmable toolkit for debugging and performance tuning parallel programs. This guide presents Moviola by describing its use as a standalone program and as a component of the toolkit. In addition, we describe the interface seen by a programmer of the toolkit.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1 Introduction

Moviola is an interactive browser used to create, examine, and manipulate graphical representations of synchronization histories of concurrent programs. It is part of an integrated, programmable kit of tools under development by the "Parallel Program Understanding Techniques and Tools" (PPUTTS) group in the University of Rochester Computer Science Department [Fowler *et al.*, 1988]. The PUTTS Toolkit is a collection of programs designed to help programmers understand in detail the behavior of parallel programs that use explicit and potentially fine-grained synchronization and locking operations to control access to shared resources. The goal is to facilitate the logical debugging, the performance debugging, and the performance analysis of these programs in much the same way interactive debuggers and profilers are used to analyze the behavior of sequential programs. The Toolkit is based on an extension of the Instant Replay [LeBlanc and Mellor-Crummey, 1987] technique for recording synchronization histories of parallel programs. Data recorded in the histories allow the deterministic replay of the program execution under a debugger as well as detailed performance analysis for debugging and tuning. *Moviola* is the common user interface for the analysis and graphical manipulation of those histories. These core facilities form a foundation upon which we are constructing more complex tools such as symbolic debuggers, execution profilers, and performance analyzers.

The synchronization history of an execution of a parallel program is a partial ordering of the events in that execution. *Moviola* represents it as a directed acyclic graph. The vertices of the graph are *events*, each of which is the execution of an operation on a shared *object* through which processes can synchronize and communicate. Instrumented synchronization primitives record the details of each operation in the local synchronization history of the invoking process. *Moviola* combines the local histories to form the global history. Depending on the style of parallel programming used in the target program, events may consist of the sending and receiving of messages, the reading and writing of shared variables protected with a locking protocol, the operations of other constructs for concurrent programming, or a combination of several of these. Each directed edge in the graph represents a temporal dependency between the pair of events it joins [Lamport, 1978]. As in Lamport's treatment of time in distributed systems, the events on each processor are totally ordered with an edge from each event to the next succeeding event on that processor. The event at which a message is sent will be joined with the event at which it is received. An edge can also represent a *conflict* ([Bernstein *et al.*, 1987], Chapter 2) dependency between operations on a shared variable. This can be a write-read dependency that arises between the writing of a value in a variable and a later operation that reads that value, a read-write conflict between a read and a subsequent write operation that destroys the value read, or a conflict between a pair of write operations.

Moviola presents synchronization history graphs as time-space diagrams (See Figure 1). In the diagram, time flows from top to bottom: all edges in the graph are implicitly directed from top to bottom. Events that occur within a single process are aligned vertically, forming a time line for that

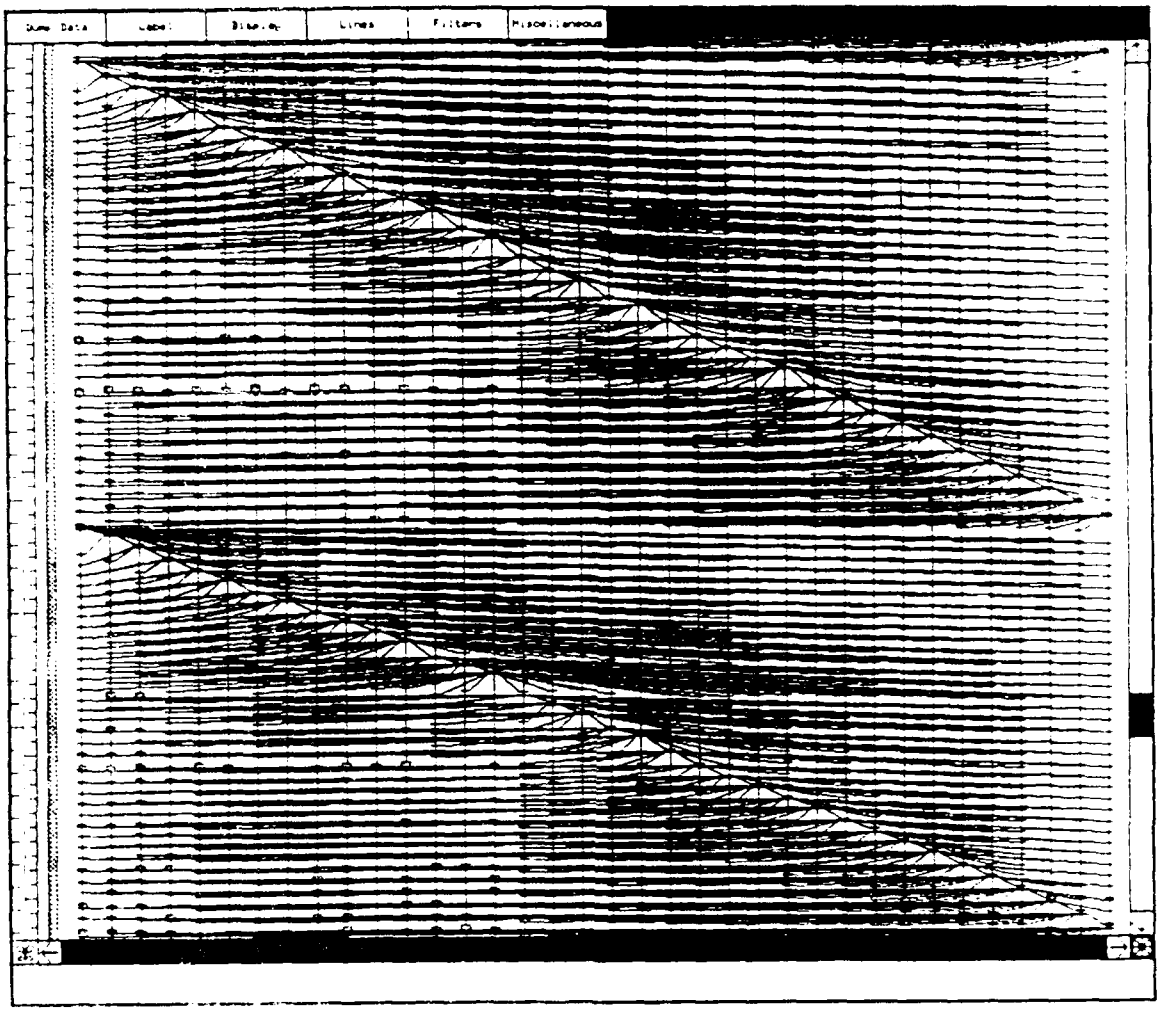

Figure 1: A Unix/Motif *Mentat* window displaying part of the synchronization history of a program that solves a systems of linear equations using parallel Gaussian elimination. The synchronization history of the program is displayed as a directed acyclic graph in the central pane. Menu headers are arrayed across the top of the graph pane. On the bottom is a message (text) pane. A horizontal elevator bar is displayed between the message and graph panes. On the right is a vertical elevator bar. At the ends of the elevator bars are arrowheads for scrolling. The small pane in the lower right hand corner contains four diagonal arrowheads. The bar on the left displays the time scale, and the small pane below that is used to "reonify" the window.

process. Diagonal edges represent inter-process dependencies. Each event is presented as a shaded box whose height is proportional to its duration. The box is divided into a waiting time component and an execution time component. Depending on the synchronization and communication primitives used, a processor may have to wait for a message to arrive, for a buffer to be filled or emptied, or for a lock on a shared variable to become available. Excessive waiting is an indication of performance problems. The graphical presentation of waiting in the *Moviola* display helps to draw the user's attention to these problems.

Although the execution of each pair of conflicting operations adds an ordering constraint between them, programmers are often concerned only with the subset of edges that entail the flow of information between processes. *Moviola* therefore uses the full set of edges to derive a consistent global clock used to determine the placement of events along the time axis, but the programmer can specify a different set to be "interesting" enough to be displayed.

Moviola can either be run as a standalone program or as part of the PPUTTs Toolkit. The Toolkit (See Figure 2.) consists of a collection of programs (tools) that run under the *aegis* of and interact through a Common Lisp system (Kyoto Common Lisp [Yuasa and Hagiya]). User interaction is through version 11 of X Windows [Gettys and Scheifler, 1986]. Tools can be written in other languages as well as in Lisp. The Lisp interface to *Moviola* includes a package through which Lisp code can access and manipulate *Moviola*'s internal data structures. This package includes functions for the management of multiple execution histories in multiple windows and facilities for extending *Moviola*'s user interface. The Lisp interface is the foundation upon which we are constructing the interfaces between *Moviola* and other components of the Toolkit. Performance analysis and debugging tools are able to install themselves to use *Moviola* both as a common execution graph manager as well as to provide a common user interface.

2 Using Moviola in Standalone Mode

To run *Moviola* in standalone mode on a Sun 3 execute:

moviola [history_directory] [-d defaults_file] [-D display_name]

The arguments to the command line are:

history directory is a path to the directory containing the synchronization history of interest. The history is stored as a set of individual process history files. The directory must contain a file named "name" whose first line is a text string specifying the prefix of the data file names. The second line of text is the name identifying which instrumentation package was used for the current synchronization history. The format of a data file name is: *prefix.poid* where *prefix* is the prefix mentioned above, and *poid* is a hexidecimal process identifier number.

-d: This option allows a .moviolarc initialization file to be specified explicitly.

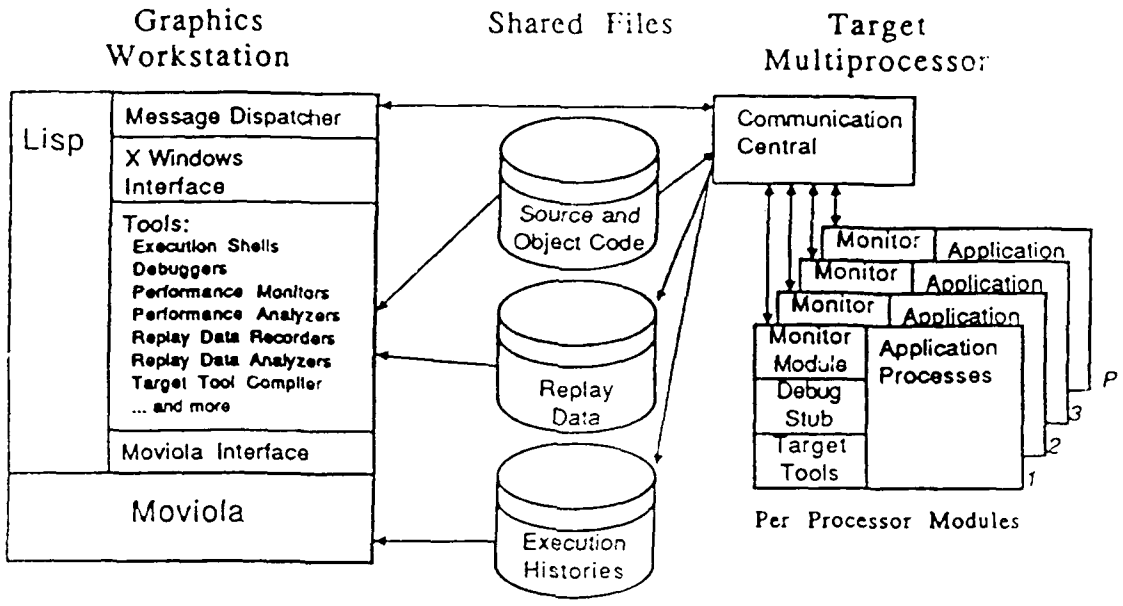

Figure 2: The organization of the PPUTTs Toolkit.

-D: The default display used by X11 is specified by the environment variable "DISPLAY". *Moviola* will use this unless otherwise specified by this option. The display name is of the format: *hostname:number.screen_number*. See the X11 documentation for a description of the display variable.

When *Moviola* starts, it opens its main window, loads the execution history (if specified), and displays the history in the graph pane (See Figure 1). The menu headers at the top of the main window are used to activate a set of pull-down menus. The darkened parts of elevator bars on the right and across the bottom show the position of the viewport in the graph pane relative to the whole history. Clicking the mouse in an elevator will move the viewport position. Clicking on an arrowhead at the ends of an elevator bar will move the viewport a fixed distance in the indicated direction. The ruler on the left displays the time scale in units of "ticks," the resolution of the clock used to record event timestamps. Messages to the user are displayed in the text window at the bottom. Text entered by the user is also echoed there. Clicking on the small pane containing a butterfly icon below the ruler will "iconify" the main window.

2.1 Interacting with *Moviola*

The graphical interface provided by *Moviola* is extremely flexible. In addition to generic panning and zooming facilities, it provides facilities for interactively customizing the user's view of the graph to focus on the interesting parts of the history. The user has the ability to define subsets of processes to display or highlight, control over the order of processes in the display, and the ability to highlight

or suppress events representing operations on specified subsets of shared resources. There is also a facility to define the set of interesting event dependencies that should be displayed.

Commands and options can be invoked through pull-down menus, pop-up menus, or through mouse events caused by pressing or clicking mouse buttons, optionally holding down one or more keys on the keyboard.

To activate a pull-down menu, point to the menu header with the mouse and hold down a mouse button. Pop-up menus are activated by mouse events while pointing to something inside the graph pane. In both cases dragging the mouse downwards will highlight each menu item as the pointer passes through it. To select a highlighted item release the mouse button. Unless otherwise stated, selecting a menu item toggles the corresponding option.

2.1.1 Moving around the synchronization graph

Moviola provides many ways to select the portion of the synchronization history graph to display in the viewport of the graph pane. Most of them are bound to mouse events. The actual button/key combinations are specified in an initialization file called ".movbindrc". For details see section 2.2.1.

Arrowheads: Clicking the mouse when the cursor is in one of the arrowheads at the end of the elevator bars will move the viewport in the indicated direction. Holding the button down will repeat the motion.

Elevator Bars: The total length of the vertical (horizontal) elevator bar represents the vertical (respectively horizontal) extent of the history graph. The dark region in each bar denotes that part of graph that is currently visible in the viewport. Clicking the cursor in either of the elevator bars will center the viewport on that relative position in the graph.

Zoom In: One can zoom in on a section of the history by designating a rectangular region of the history to be expanded to fill the graph window. The x and y coordinates are scaled independently. The region is designated by selecting one of its corners with the mouse, and while the appropriate key/button combination (by default, (no keys)/middle button) is held down, drag the cursor to the diagonally opposite corner and release.

Zoom Out: The contents of the current graph pane are scaled down to fit into a rectangular area. The area is designated as described above. The default key/button combination is (no keys)/right button.

Jump: By clicking the button/key combination (by default (no keys)/left button) in the graph pane, the point where the mouse was will be moved into the center of the graph pane.

Scroll: Holding this button/key combination (by default shift/left button) down changes the cursor to a hand that "grabs" the graph so that it can be moved around. Releasing the cursor will leave the graph in the new position.

Undo: By clicking this button/key combination (by default control/middle button) in the graph pane, you can undo the effects of the last operation in this section.

Ruler: Selecting a time interval by selecting and dragging with the left mouse button over a portion of the ruler bar on the left side of the display opens a small window describing that interval both in terms of "ticks" and milliseconds.

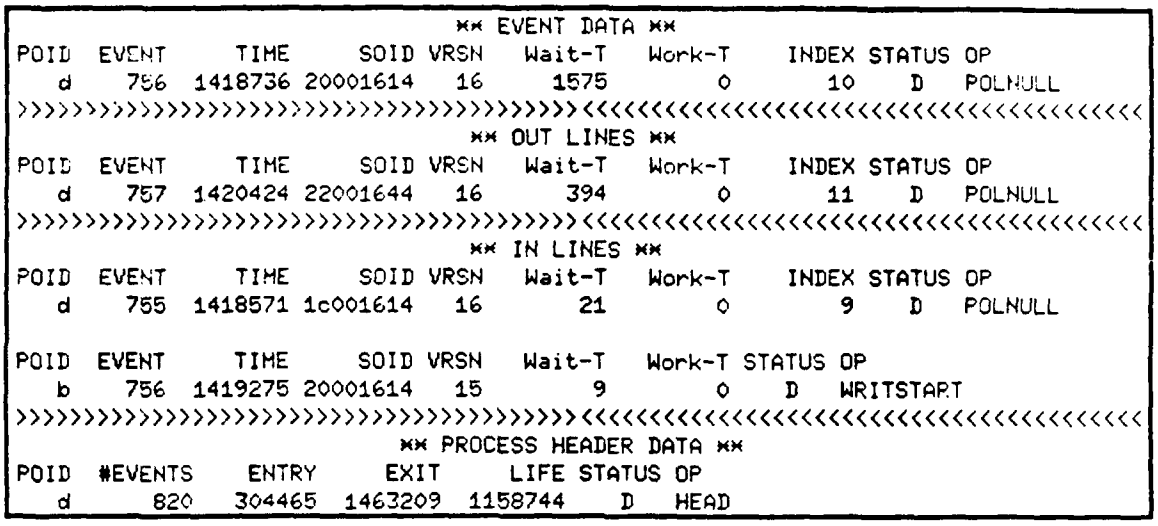

Figure 3: An event data window.

2.1.2 Examining events in detail

The 'data' command is used to open data windows in which the details of selected events are displayed. It is also used to examine details of the history as a whole. By default the key/button combination is shift/middle button. Pressing and holding this combination in the graph pane or in a data window will pop up a menu.

Pressing and holding on an event in the graph pane activates the Event Data menu pop-up menu. Pressing and holding on a line of text representing an event in a data window has a similar effect. The first item in this menu is "Event Data". Selecting this item will open a data window containing details about the event. See Figure 3. The data given in this window is determined by the first four items in the "Dump Data" pull-down menu:

In Lines: List events that directly affect this event (with their event data).

Out Lines: List events (with their event data) that this event directly affects.

Proc Data: Display raw data from the header of the process containing this event.

Event Data: Display details about this event.

Pressing and holding in the graph window, but not on an event, opens a History Data menu. The first item in this menu will always be "History Data". Selecting this item will display the data about the history in the message window at the bottom of the *Moviola* window.

In addition to the Event Data and History Data items, the installation of other tools in the Toolkit can extend the *Moviola* user interface by adding additional items to these menus.

2.1.3 Controlling the Display

Four of the pull-down menus contain commands and options that control the appearance of the graph in the display.

Label Menu: This menu controls the labelling of events in the display. The options are mutually exclusive.

Op ID: Display the identifier for the type of event.

Obj ID: Display the ID of the object referenced by the event

Proc ID: Display the ID of the process containing this event.

Event ID: Display the ID of the event within its process.

Display Menu: This menu controls the major display modes: logical vs. physical time base, display dependencies from .movsyncrc or .moviolarc, and specify the display order of the processors.

Logical: Toggle between physical and logical time bases. If the physical time base is chosen, the time axis is a good approximation to a consistant global clock to within the granularity of the local timestamps recorded in events. The logical time base is a topological sort based on the dependencies defined in the .movsyncrc defaults file. The logical time of an event is the layer of a topological sort of the graph in which it appears. No reference is made to the time stamps in events, hence all of the events have the same height.

Sync Display: Toggle between displaying the dependencies defined in .movsyncrc and those defined in .moviolarc. See sections 2.2.2 and 2.2.5. This option also affects the dependencies listed in Event Data windows.

Process Order: Selecting this item will open a window that displays the current process order. The process at the top of the list is the leftmost process in the display, and the process at the bottom of the list is the rightmost. There are three ways to modify the ordering. Clicking on ">" or "<" in this window will order the processes in descending (respectively, ascending) process identifier numbers. Clicking on "H" will order the processes by a heuristic that attempts to reduce the number of edge crossings in the display. The heuristic ordering places the process with the most edges in the center. The processes with the most edges connected to the already placed process(es) are placed on either side. This step is iterated until all of the processes have been placed. Manual reordering is the third method. Pressing and holding the mouse on one of a process identifier in the window, and then releasing the mouse on top of another process identifier will move the first process past the second in the direction moved. Moving a process upwards places it before the second process; moving it downward places it after the second.

Lines Menu: The first three items of this menu control the display of cross edges between events on the display and those off the screen. If none of the first three options are chosen, then only the edges between displayed events appear.

In Lines: For any event currently in the graph pane, display the edges coming onto the screen from events that directly affect this event.

Out Lines: For any event currently in the graph pane, display the edges leaving the screen to events that this event directly affects.

All Lines: Display all edges of the graph that intersect the screen, whether or not they are incident to events or the screen. Since this option slows the display time, and clutters the graph pane, it is normally not used interactively with *Moviola*. Its main use is to insure consistency when a hard copy is constructed from multiple screen dumps. Choosing this item deselects both of the previous two items.

OnlyDirect: Display only those cross edges for which the second event waits for the first.

Filters Menu: Whether an event or edge is displayed and whether it is highlighted are controlled by a number of filters. An event or edge is displayed only if all filters allow it to be. An event or edge is highlighted if highlighting is requested by at least one filter. Which subset of process, object, and operation-type filters to use and whether they affect events and/or edges are determined by activating the first selection under this menu.

Choose Filters: This option opens a window for choosing which filters are used and whether they are used for events and/or cross lines. The possible filters are "Process", "Object", and "OpID". The filters are defined by partitioning the set of processes or objects into three categories: not displayed, displayed, and highlighted.

Choose Process: This option opens a window for defining the process filters. In the window, clicking on the row of a process ID will give that process the status corresponding to column you clicked on. If you click on "Display All", "Display None", or "Highlight All" in the window, all processes will get the corresponding status. Choosing this item when the window is already open will close the window. The process order is recomputed and the graph is redisplayed when the window is closed or when you click on "redisplay" in the window.

Choose Object: This option opens a window for defining the object filters. In the window, clicking on the row of a object ID will give that object the status corresponding to column you clicked on. If you click on "Display All", "Display None", or "Highlight All" in the window, all of the objects will get the corresponding status. Choosing the menu item when the window is already open will close the window. The graph is redisplayed when the window is closed or when you click on "redisplay" in the window.

Choose OpID: This option opens a window for defining operation identifier filters. In the window, clicking on the row of a the identifier of an operation type changes the operation type to the status corresponding to column you clicked on. If you click on "Display All", "Display None", or "Highlight All" in the window, all operation types will get the corresponding status. Choosing the item when the window is already open will close the window. The graph is redisplayed when the window is closed or when you click on "redisplay" in the window.

2.1.4 Miscellaneous Commands Menu

Miscellaneous Pane: The "Miscellaneous" menu provides the means to obtain help as well as to perform miscellaneous operations such as reading an alternate .moviolarc defaults file, reading a new synchronization history, and redrawing the screen. Information about the defaults files are explained in the Defaults File section of the "man page".

Man Page: This item opens a text window with the manual page. If chosen while the help window is open, the window is closed.

Bindings: This item opens a window showing the current bindings of mouse button/key combinations to commands.

ReDraw: This item will refresh everything in the main window.

New File: This item is for getting a new history file attached to this window. A request for the history directory name will appear in the message window. Type the name followed by return to enter it.

Defaults: This item reads a new .moviolarc defaults file. The file name is requested the same way as the New File option.

Raise Data: This item will raise all of the data windows.

New Window: This item will open another copy of *Moviola* .

Sync Clocks: This item will synchronize the clocks using the dependencies defined in the .movsyncrc file.

The Close Function: This exits *Moviola* . A button/key combination is also bound to a "CLOSE" command that is defined for all *Moviola* windows. Before any window is actually closed you must confirm that you really intended to by performing a second button click in the same window.

Keyboard If the mouse pointer is in the main window of the *Moviola* package, any keystroke will be sent to the message window. Pressing return will send the string to be processed. This is used for entering file names to choose a new defaults file, a new display, or a new file. If you type "quit" and press return, *Moviola* will quit.

In All Text Windows: If a text window has "<<<MORE >>>" at the bottom, then a button click (except for any button/key combination defined to do something else) in the window will get the next page of text.

Clock Data: Under the "Dump Data" menu is a item labeled "Clock Data". This item opens a window that displays the relations between the local clocks of any pair of processes. The left button goes forward through the list of pairs, the right button goes backward. The middle button zooms in and out by clicking. The shift/control/or meta keys along with the left button toggles the process-1 to process-2 dependency points (the squares). One of the keys with the left button toggles the process-2 to process-1 dependency points. One of the keys with the middle button toggles the displaying of lines. The lines represent the the synchronization where the slope is the scale, and the y intercept is the offset of process-2 in relation to process-1.

2.2 Customizing *Moviola*

When *Moviola* is started, it configures itself according to the contents of three initialization files. All initialization files have a common syntax consisting of command lines, each of which is a sequence of keywords and values. All words must be separated by white-space, defined to be any sequence of the following characters: [space], '=', ':', '&', '|', "'", and '"'. In each section the separators are chosen by convention for readability. Whenever the character '#' is found in a file, the rest of the line is treated as a comment. Each command line is terminated by the end of the line. All information in initialization files is case insensitive.

Each of the three initialization files has a distinct function. The file ".movbindrc" binds commands to keys and buttons and defines the initial window configuration. The initialization file .movsyncrc.<package-name > defines the inherent temporal dependencies among events as determined by the instumentation library that records the history. These are generally a superset of the dependencies a user wants to display. This file is usually created by the authors of the synchronization library and is not modified by the user. The file .moviolarc.<package-name > defines the initial status of the display and the set of user-defined dependencies to display.

An instrumentation package designator must be part of the names of the ".moviolarc" and ".movsyncrc" files to designate the instrumentation that recorded the history. For example, ".moviolarc.chrys" indicates that our standard Chrysalis instrumentation was used.

2.2.1 .movbindrc

This file defines bindings between mouse events (button/key combinations) to *Moviola* commands. Mouse events that are not defined here may inherit commands from the X window manager. The

```
#PARAMETER = INITIAL VALUE
#--------- = -------------
X          =  10
Y          =  200
WIDTH      =  512
HEIGHT     =  512
#
#FUNCTION  =  KEYS      :  BUTTON
#--------  =  ----      :  ------
CLOSE      = SHIFT      : RIGHT
DATA       = SHIFT      : MIDDLE
SCROLL     = SHIFT      : LEFT
JUMP       = NONE       : LEFT
ZOOMOUT    = NONE       : RIGHT
ZOOMIN     = NONE       : MIDDLE
UNDO       = CONTROL    : MIDDLE
```

Figure 4: The default .movbindrc file.

file also specifies the initial placement of the main window. The current directory, the user's home directory, and then a system-defined standard directory are searched in that order. It is read only when *Moviola* is initialized. Figure 4 illustrates the definition of the default bindings.

Binding Definitions The format used to bind a command is *function = keys : button*. (Note that the separators used here are chosen by convention.) This is just like the binding commands in a uwmrc file (used by the uwm window manager) except that the context is always "window" (as opposed to "icon"). Holding down the specified keys on the keyboard while clicking or holding the mouse button will invoke the command.

KEYS: The choices are SHIFT, META, CONTROL, ALL, and NONE or any combination of the first three (i.e. "SHIFT & META", etc.).

BUTTON: The choices are RIGHT, MIDDLE, and LEFT.

The *Moviola* commands that can be bound to mouse events are:

ZOOMIN: This is the "zoom in" command defined on the graph pane.

ZOOMOUT: This is the "zoom out" command defined on the graph pane.

CLOSE: This is the "close" command defined on all *Moviola* windows.

DATA: This is the command that will bring up the pop-up menus in the graph pane.

JUMP: This is the "jump" command defined on the graph pane and data windows.

SCROLL: This is the "scroll" command defined on the graph pane.

UNDO: This is the "undo" command defined on the graph pane.

Initial Configuration This section defines the initial placement and size of the window. Each command line takes the form

parameter = initial value.

X & Y: These are to define the initial x and y coordinates (in pixels) relative to the root window.

WIDTH & HEIGHT: These are to define the initial width and height (in pixels) of the tool's window.

2.2.2 .movsyncrc

This file specifies when a pair of events on a shared object define an inherent temporal dependency. This is determined by the semantics of the instrumented synchronization primitives used to record histories. For that reason, a .movsyncrc file is usually created by the author of the corresponding synchronization packages. *Moviola* only assumes that the timestamps each process uses in recording its own history are generated by a local clock. The inherent temporal dependencies are used as the basis for deriving a single consistent gobal time base. We use the method described in [Duda *et al.* 1987] to derive our best approximation to a global physical clock. The .movsyncrc file is read once when *Moviola* starts up. The search path is the data directory, the current directory, the user's home directory, and finally the standard directory.

Class Definitions An instumentation package assigns an integer operation type code to each type of event. This classification is usually finer than needed for deriving a consistent global clock. Class definitions are therefore used to aggregate operation types into coarser equivalence classes. The command *optype* = *class* assigns the operation type to the corresponding class. *Optype* is either an integer operation code defined in the instrumentation package or a keyword denoting one of the following following system-defined event types: MASTER_PROCESS, PROCESS_HEAD, USER_DEFINED_TAG, SYSTEM_DEFINED_TAG, PROCESS_CREATE, EVENT_ERROR, or DIVISION. Note that the *class* number cannot be larger than the maximum *optype* number plus 7 (for the 7 system types). A typical set of operation types is defined in the sample .moviolarc file.

Dependency Definitions This section defines predicates that specify whether an event is considered dependent on another event. Dependency is determined by applying a test to pairs of events in the classes defined in the previous section. The format of a dependency definition is *cross*: *1stField* : *rel* : *2ndField*. *Cross* is of the form *class*->*class*. An event of the 2nd *class* depends on an event of the 1st *class* if both events are operations on the same object, and the value of the *1stField* of the first event is in relation *rel* to the value of the *2ndField* of the second event. The possible *Field*'s are POID, OPID, VRSN, ENTRY, EXIT, LIFE, WTIME, and ETIME. These fields are recorded by all history recorders. *Rel* must be one of <, >, *N*<, or *N*>. *N*< and *N*> mean that the *1stField* must be *N* less than or greater than the *2ndField* respectively. The string "==" has been specially defined as a *REL* since the character '=' is considered white-space. For example, "2->3 VRSN 1<VRSN" means "events of class 3 depend on events of class 2 when the version number of the first event is one less than the version number of the second event." (Note that "->" is NOT white-space, and there is no white-space between the two classes.) If two event classes always depend on each other, then NONE can be used instead of the *FIELD*'s and *REL*. The only tests currently used are comparisons of object version numbers.

2.2.3 .moviolarc

This file has four sections. The first specifies the initial state of the display. The second defines filters to be used on events and cross lines, the initial process and object filters, and the initial ordering of the processes. The third section defines a set of operation classes used in the display and it specifies how events are to be labelled. The fourth part defines the set of dependencies to be displayed. A

```
#
#INDEX =   TYPE  #  LABEL
#----- =   ----  #  -----
0      =   2     #  PollWriteStart
1      =   3     #  PollReadStart
2      =   4     #  PollNull
3      =   2     #  MemoryDelete
4      =   8     #  EventReset
5      =   8     #  EventPost
6      =   8     #  EventData
7      =   8     #  EventDelete
8      =   8     #  EventWait
9      =   8     #  EventMWait
10     =   5     #  DualQEnq
11     =   5     #  DualQTryEnq
12     =   5     #  DualQWait
13     =   5     #  DualQPoll
14     =   5     #  DualQDelete
15     =   3     #  MemoryReadStart
16     =   7     #  MemoryReadEnd
17     =   2     #  MemoryWriteStart
18     =   6     #  MemoryWriteEnd
MASTER_PROCESS   =   1
PROCESS_HEAD     =   1
PROCESS_CREATE   =   0
#
#CROSS      :   1st FIELD :   TEST  :   2nd FIELD
#-----      :   --------- :   ----  :   ---------
#PC->H
0->1       :   NONE
#WE->RS
6->3       :   VRSN      :   ==    :   VRSN
#RE->WS
7->2       :   VRSN      :   ==    :   VRSN
#WE->WS
6->2       :   VRSN      :   ==    :   VRSN
#WE->P
6->4       :   VRSN      :   ==    :   VRSN
#DQ->DQ
5->5       :   VRSN      :   1<    :   VRSN
#EV->EV
8->8       :   VRSN      :   1<    :   VRSN
```

Figure 5: The default file: .movsyncrc.chyrs. This is the default file corresponding to the standard set of synchronization primitives used with programs running directly on the Chrysalis operating system.

.moviolarc file is read every time a history is loaded. *Moriola* searches first in the directory from which the history is read, then the current working directory, and finally the user's home directory. This search path is overridden if the -d option is used on the command line. A new .moviolarc file can also be read explicitly using the "Defaults" command under the "Miscellaneous" menu. If a dependency creates an edge that appears to go backwards in time an error message is printed and the edge is discarded.

Initial Display This section specifies the initial state of the display. The format of commands is *variable* = *initialValue*. Refer to the sample default file for reference.

XSTART & YSTART: These are the coordinates of the point that is initially displayed in the upper left hand corner of the graph pane. If XSTART = N then the Nth process of the display will be placed in the left side of the graph pane. If YSTART = T and the physical time base is chosen, then time T (in ticks) will be placed at the top of the window. If the logical time base is chosen, then if YSTART = T, the Tth layer of the graph will be placed at the top.

XSCALE & YSCALE: These are the initial scale factors used for the graph in the x and y directions. The x dimension is measured in processes, and the y dimension is measured in ticks or levels (for the logical display).

DUMPDATA: This initializes the menu specifying what raw data displayed when an event is opened. The initial value is specified by the keywords: EVENT, PROCESS, OUT, and IN.

LABEL: This initializes the menu that specifies the type of label used for events in the graph pane. The initial value is specified by the keywords: OPERATION, PROCESS, OBJECT, and EVENT.

LOGICAL: This initializes the menu item controlling whether the logical or physical time base is used. Possible values are ON and OFF.

SYNC: This initializes the menu item that specifies whether to display the dependencies from .movsyncrc rather than those defined in .moviolarc. Possible values are ON and OFF.

LINE: This initializes the menu that specifies which lines are being displayed. Possible values are IN and/or OUT, or ALL.

ONLYDIRECT: The option can be initialized as ON or OFF.

Filters This group of commands can enable and initialize the commands available under the "Filters" menu. A line of the form

entity : FILTER : *object*,

where *entity* is one of PROCESS, OBJECT, or OPERATION and *object* is one or more of EVENT or LINE will enable application of the filter for the entities to the specified class of graphical objects. (Note that filtering by operation type is applicable only to events.)

The filters are initialized by a sequence of lines of the form

entity : *status* : { *identifier mid* ALL },

where *entity* is one of PROCESS, OBJECT, or OPERATION, *status* is one of HIGHLIGHT, DISPLAY, NODISPLAY, or REMOVE, and *identifier* is the numerical identifier of the entity affected: a process identifier, an event type number, or an operation type number. The keywords NODISPLAY and REMOVE are synonymous. Since the lines are processed in order the easy way to suppress the display of a single process is to first request that all processes be displayed and then REMOVE the appropriate process.

The initial ordering of processes is defined by a line of the form

```
#VARIABLE = INITIAL CONSTANT[s]
XSTART   = 0
YSTART   = 0
XSCALE   = 50
YSCALE   = 20
DUMPDATA = EVENT & OUT & PROCESS
LABEL    = OPERATION
LOGICAL  = OFF
SYNC     = OFF
LINE     = IN & OUT
ONLYDIRECT = OFF
#
#PROC/OBJ : DIS/NODIS/REM/HL : ALL/OBJECTID
#    /LISP: FILTER     : EVENT &| LINE
#         : ORDER      : <,>,HEURISTIC
#-------- : --------   : -----------
PROCESS   : DISPLAY    : ALL
OBJECT    : DISPLAY    : ALL
OPERATION : DISPLAY    : ALL
PROCESS   : FILTER     : EVENT & LINE
OBJECT    : FILTER     : EVENT
OPERATION : FILTER     : EVENT
PROCESS   : ORDER      : <
#
#INDEX =   LABEL          |  ABBR.   |  TYPE
#-----  =   -----          |  -----   |  ----
0       =   PolWriteS     |  PWS     |  1
1       =   PolReadS      |  PRS     |  2
2       =   PolNull       |  PN      |  5
3       =   MemoryDel     |  MD      |
4       =   EvntReset     |  ER      |  8
5       =   EvntPost      |  EP      |  8
6       =   EvntData      |  EDt     |  8
7       =   EvntDel       |  ED      |  8
8       =   EvntWait      |  EW      |  8
9       =   EvntMWait     |  EM      |  8
10      =   DualqEnq      |  DE      |  7
11      =   DualqTry      |  DT      |  7
12      =   DualqWait     |  DW      |  7
13      =   DualqPoll     |  DP      |  7
14      =   DualqDel      |  DD      |  7
15      =   ReadStart     |  RS      |  2
16      =   ReadEnd       |  RE      |
17      =   WritStart     |  WS      |  1
18      =   WriteEnd      |  WE      |  6
MASTER_PROCESS     =   Master       |  M       |  0
PROCESS_HEAD       =   Head         |  H       |  0
USER_DEFINED_TAG   =   UserDfTag    |  UT      |
SYSTEM_DEFINED_TAG =   SysDfTag     |  ST      |
PROCESS_CREATE     =   ProCreate    |  PC      |  3
EVENT_ERROR        =   ERROR        |  E       |
DIVISION           =   DIVISION     |  DV      |  4
#
#CROSS      :   1st FIELD :   TEST  :   2nd FIELD
#-----      :   --------- :   ----  :   ---------
1->2        :   VRSN      :   1<    :   VRSN   #W->R
#2->1       :   VRSN      :   \=    :   VRSN   #R->W
#1->1       :   VRSN      :   1<    :   VRSN   #W->W
3->0        :   NONE                           #PC->H
#4->4       :   VRSN      :   1<    :   VRSN   #DV->DV
1->5        :   VRSN      :   1<    :   VRSN   #W->P
7->7        :   VRSN      :   1<    :   VRSN   #DQ->DQ
8->8        :   VRSN      :   1<    :   VRSN   #EV->EV
```

Figure 6: A sample .moviolarc.chrys file: This is the default file corresponding to the standard set of synchronization primitives used with programs running directly on the Chrysalis operating system.

PROCESS : ORDER : {'<' | '>' | HEURISTIC }.

Labels This section defines equivalence classes on operation types and assigns display labels to them. Each line is of the form

optype : *label* | *abbr* | *class*

The *label* and *abbr* are the name and abbreviation to display on an event if labelling by operation identifier is requested. *Optype* and *class* are used the same way they were used in .movsyncrc.

Dependency Abstraction This section defines the subset of dependencies actually displayed in the graph. It uses the classes defined in this file, and the format is the same as in .movsyncrc.

3 Using *Moviola* as part of the PPUTTs Toolkit

The graphical display capabilities of *Moviola* make it a useful tool for analyzing the correctness and performance of a concurrent program through the observation in detail of synchronization and *communication* behavior. Despite this utility, we want and need additional functionality beyond the graphical manipulation facilites we have described thus far. Source language debuggers, statistical analysis tools (including profilers), and critical path analysis are potential extensions that we might want to make directly to *Moviola* . The set of extensions, however, is not limited to these few. Each source language, each target machine, and many application programs will need individually customized extensions. Furthermore, the sheer size of some execution graphs and the drudgery of traversing them by hand will cause some users to want to make *ad hoc* extensions in response to phenomena seen in a particular execution graph.

To satisfy these needs it is necessary that *Moviola* be made both dynamically extensible and programmable. The PUTTs Toolkit provides these properties by running *Moviola* and other parallel program analysis tools under a Lisp system. Running *Moviola* in this mode provides extended functionality both by allowing one to use a library of existing analysis tools, to write one's own extensions, and to interactively program *ad hoc* analyses.

3.1 Starting *Moviola* under the Toolkit

The first step is to start the Toolkit. See the online manual page for *pputts* to get started. *Pputts* is a modified version of Kyoto Common Lisp. When it starts up you are interacting with the Lisp interpreter. To obtain the data needed for finding the other parts of the Toolkit, *pputts* reads a file accessed by the path "../.tools". To list the available PPUTT's tools execute the Lisp form (**pputts-list-tools**). The function (**pputts-load** *toolname*) will load and initialize one of the listed tools. For example, (**pputts-load 'moviola**) loads *Moviola* . Once *Moviola* has been loaded the simplest method of starting it is to execute the form (**moviola-start &optional** *hist-dir* **:display**

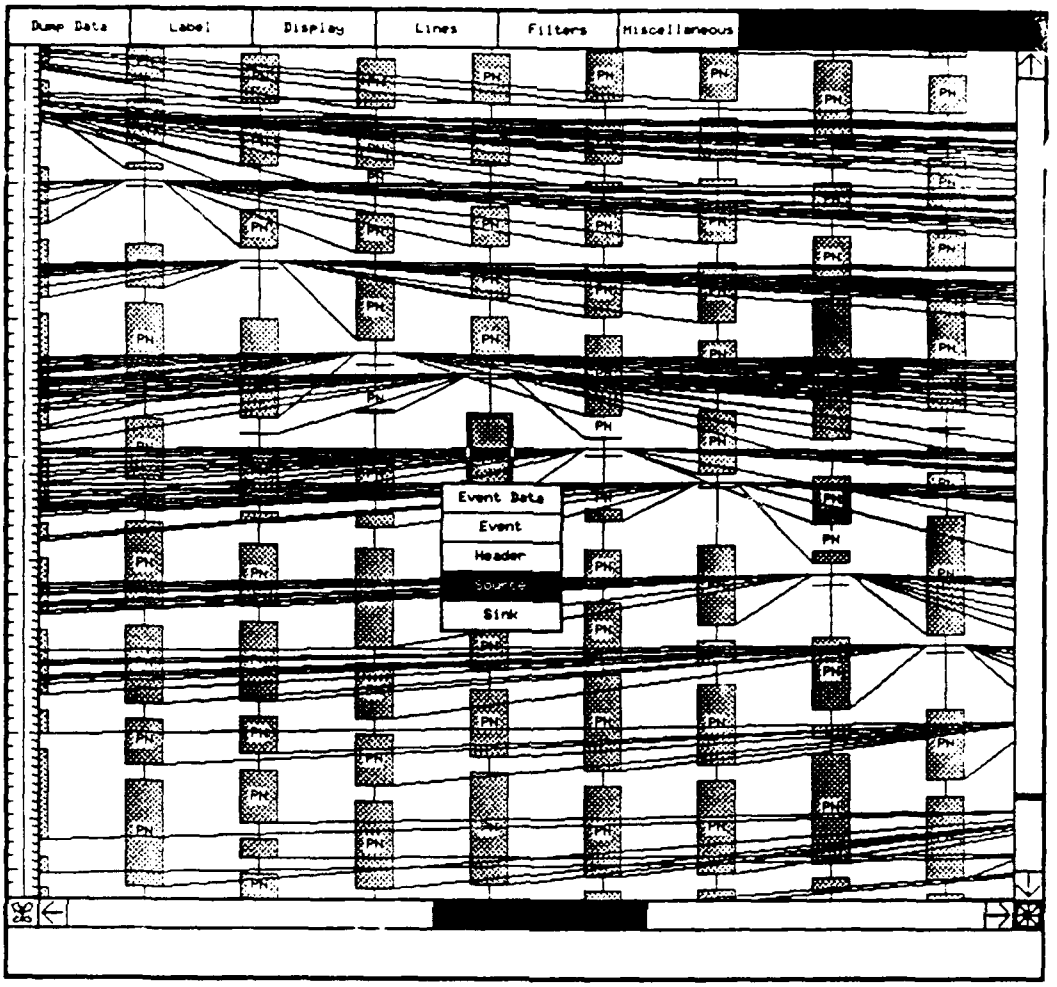

Figure 7: Selecting a Source Node

name :rc *rcname*). This is equivalent to starting *Moviola* from the command line and has the same arguments. (Note that ":display" and ":rc" are keywords for the optional arguments that follow.) A *Moviola* window is opened and you can interact with it exactly as you did in standalone mode.

Moviola 's functionality is extended by loading packages of analysis tools. For example, to load the standard waiting-statistics tool, execute **(pputts-load 'waiting)**. The following functions tabulate the waiting time of an execution by process or by object.

(all-process-wait-total *history*) ->list: This function returns a table in list form of total waiting time tabulated by operation for each process.

(all-object-wait-total *history*) ->list: This function returns a table in list form of total waiting time tabulated by operation for each object.

3.2 Critical Path Analysis

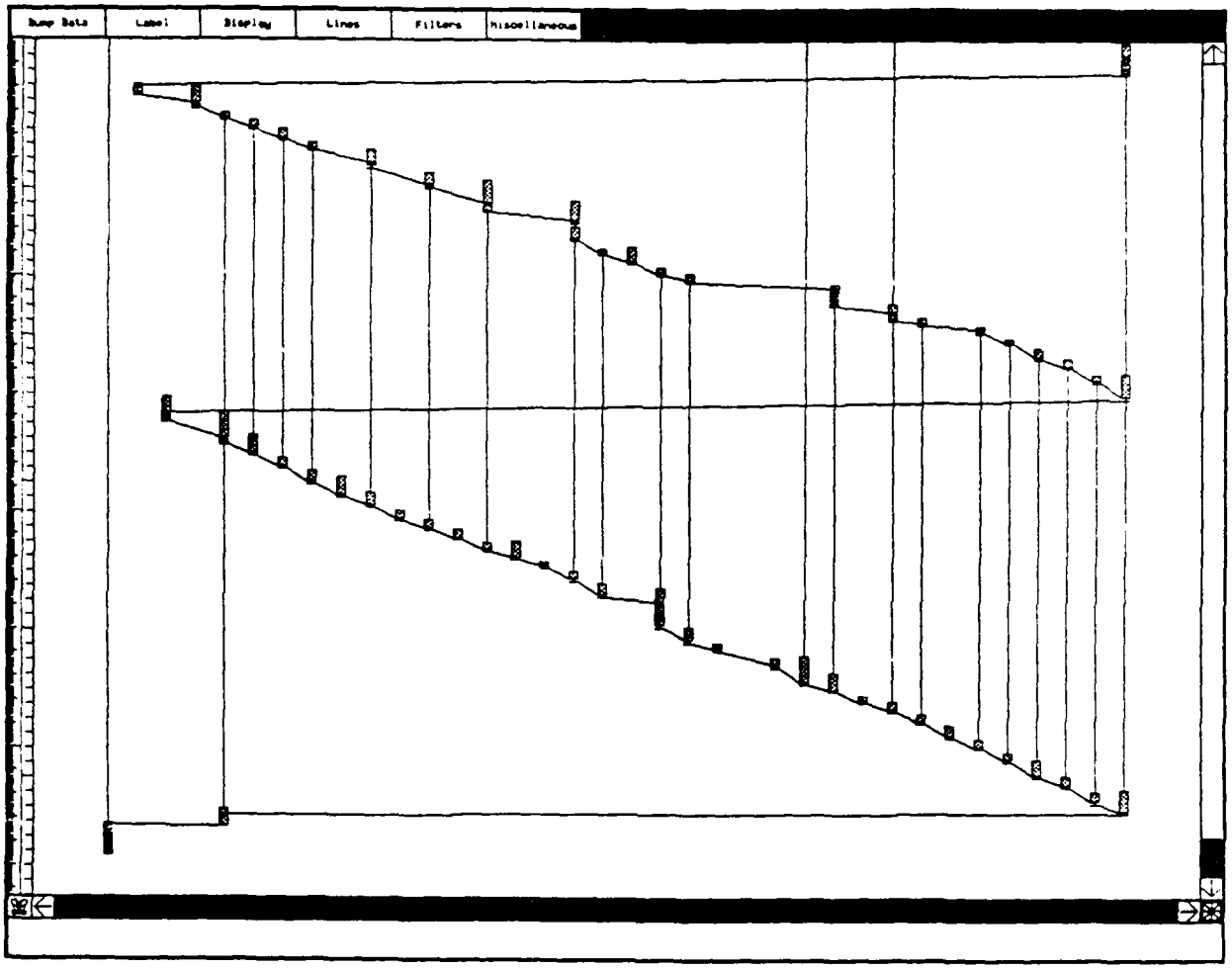

Figure 8: Highlighted Critical Path

Another useful tool performs critical path analysis. To load this tool execute (**pputts-load 'critical**). The package defines the following functions and variables

critical-path: To compute the critical path from a source event to a sink event execute (**critical-path** *source sink*), where *source* and *sink* are events. The list of events returned is the critical path. Executing (**main-critical-path** *history*) will compute and return the critical path from the first to the last event of the history.

critical-env: Executing (**critical-env**) installs an interactive facility for computing and displaying critical paths. The user interface is through several items that **critical-env** adds to the event and history pop-up menus. (The utility for modifying menus utility is explained in section 4.2.3.) Two new event menu selections labeled "Source" and "Sink" are used to define the source and sink events. A new history menu selection labeled "Use Sync" toggles between using all dependencies in computing the critical path and using only those that are displayed. After selecting source and sink events (figure 7), the user chooses a new history menu selection labeled "Critical Path" to compute the critical path. The path is highlighted, the rest of the synchronization history is not displayed, and the path is associated with the history.

The history menu selection labeled "Crtcl Path 2" will calculate the critical path, and then calculate a second path which is found by setting all of the edges in the critical path to zero and then recalculating the longest path (with respect to execution time). Only the events in the two paths will be displayed, and the events in the second path that are different from the first are highlighted. Both paths are associated with the history.

The relations between the Lisp objects computed in each of these actions and the history are created and maintained by an "association utility". (See section 4.2.2.) The values of the global variables *source-id* and *sink-id* are the indices used by the association utility for storing and retrieving source and sink events. Similarly, it uses the indices *sync-id* to store the flag that specifies which set of dependencies to use, *path-id* for the critical path, and *path2-id* for the second path.

4 Programming *Moviola* with Lisp

The *Moviola* tool supplies the programmer with a set of functions that access and manipulate *Moviola*'s internal data structures. There is also a set of utilities that allow Lisp functions and variables to be accessed through the *Moviola* user interface, thus extending it. New tools are integrated with the rest of the system by using these facilities.

4.1 History functions

The following functions access and manipulate the internal *Moviola* data structures that comprise a synchronization history.

4.1.1 Sync Functions

These functions control the choice of dependency sets used for the display. If **event-sync** is turned on, the dependencies defined by the synchronization package in .movsyncrc.*package* are used rather

than the dependencies defined by the user in the file .moviolarc.*package*.

```
func: (event-sync)-> t/nil    /* The current sync status */
func: (event-sync-on)         /* Turn sync status on */
func: (event-sync-off)        /* Turn sync status off */
```

4.1.2 The Cross Line Data Structure

An **xline** structure represents a cross line between two events. For each event the first **xline** in the xline-out linked list is a link to the next event (if it exists) in the same process. Similarly the first **xline** in the xline-in linked list is a link to the previous event (if it exists) in the same process. The rest of the cross lines are the other dependencies defined in the defaults files.

```
macro: (xline-in xline) -> xline     /* Next in cross line */
macro: (xline-out xline) -> xline    /* Next out cross line */
macro: (xline-from xline) -> event   /* The event pointing */
macro: (xline-to xline) -> event     /* The event pointed to */
macro: (xline-status window event) -> *nodisplay* | *display* | *highlight*
macro: (xline-field xline) -> int
macro: (xline-field-set xline int)
```

The **field** field of the structure is reserved for the use of the Lisp programmer.

4.1.3 The Event Data Structure

An **event** structure represents an event in the synchronization history.

```
macro: (event-next event) -> event        /* Next event in process */
macro: (event-id event) -> int            /* Event id */
macro: (event-history event) -> history   /* The history of this event */
macro: (event-rel-x event) -> int         /* The relative x position */
macro: (event-rel-y event) -> int         /* The relative y position */
macro: (event-log-y event) -> int         /* The logical y position */
macro: (event-height window event) -> int /* The event height */
macro: (event-head event) -> event        /* The head event of this process */
macro: (event-head? event) -> t/nil       /* Is the event the head event? */
macro: (event-last event) -> event        /* The last event in this process */
macro: (event-out event) -> xline         /* The first out cross line */
macro: (event-in event) -> xline          /* The first in cross line */
macro: (event-status window event) -> *nodisplay*/*display*/*highlight*
macro: (event-incomplete event) -> t/nil  /* Is the event incomplete? */
macro: (event-prev event) -> event        /* The previous event in this process */
macro: (event-objectID event) -> int      /* The id of the object acted upon */
macro: (void-objectID? int) -> t/nil      /* Is the object id void? */
  /* (incomplete or division event ) */
macro: (event-vrsn event) -> int          /* Version of the object acted upon */
macro: (event-opid event) -> string       /* The operation label of the event */
```

```
macro: (event-stime event) -> int          /* The start time of the event */
macro: (event-access event) -> int         /* The access time of the event */
macro: (event-exit event) -> int           /* The exit time of the event */
macro: (event-wait-time event) -> int      /* Time the event waited to access */
macro: (event-work-time event) -> int      /* Time the event worked on object */
```

The **event** fields **field1** and **field2** are reserved for the use of the Lisp programmer.

```
macro: (event-field1 event) -> int
macro: (event-field1-set event value) -> int
macro: (event-field2 event) -> int
macro: (event-field2-set event value) -> int
```

The **event numin** field is set to be the number of "interesting" incoming edges by the function **hist-reset-incounts**. At other times this field can be used as desired by the Lisp programmer.

```
macro: (event-numin event) -> int
macro  (event-numin-set event value) -> int
macro. (event-numin-dec event) -> int      /* Decrement the numin field */

func  (center-win-event window event)   /* Center an event in a graph pane. */
```

4.1.4 The History Data Structure

These forms return global information about a history.

```
macro. (hist-number procs hist) -> int     /* The number of processes */
macro: (hist-name hist) -> string          /* The name of the program */
macro: (hist-process hist int) -> event    /* The head of the int'th process */
macro: (hist-mainproc hist) -> int         /* The head event of the main process */
macro: (hist-firstevent hist) -> event     /* The first event in the graph */
macro: (hist-lastevent hist) -> event      /* The last event in the graph */
```

Hist-reset-incounts recomputes the **event-numin** fields of all events in the history. The only edges regarded as "interesting" in this computation are those that can be traversed by paths rooted at the specified event **event**. This function is used for computing a topological sort of the graph. The **field1** and **field2** of the **event** structures are modified by this function.

```
macro: (hist-reset-incounts hist event)
```

4.2 User Interface

The following functions and variables are used to affect the current state of the display and to extend the *Moviola* user interface.

To make a new tool known to the Toolkit, execute (**pputts-save** *keyname filename* &rest *dependencies*), where *keyname* is the name by which the tool should be known, *filename* names the file in which to find it, and *dependencies* lists the tools upon which this new tool depends. Attempting to load the new tool will ensure that all the *dependencies* are also loaded.

4.2.1 *moviola-window* and *moviola-history*

These two global Lisp variables point to the most recently referenced *Moviola* window and history, respectively. If a window is referenced without being associated with a history, then ***moviola-history*** will be null. Similarly ***moviola-window*** is null if there is a reference to a history not associated with a window.

4.2.2 Association utility

The association utility maintains association lists attached to all windows and histories. To define a new field for either type of object, use the functions (**new-win-assoc** *init_function*) or (**new-hist-assoc** *init_function*). These functions return the integer index of the newly created field. *Init_function* is used to initialize the new field. It should take a window (respectively, a history) as its argument and return an initial value. It is called for every existing association list when the new field is created and it is called whenever a new association list is created. The functions (**win-assoc** *window id_number*) and (**hist-assoc** *history id_number*) return the value of a field. The functions (**change-win-assoc** *window id_number new_value*) and (**change-hist-assoc** *window id_number new_value*) set the value of a field. The functions (**remove-win-assoc** *id_number*) and (**remove-hist-assoc** *id_number*) remove fields from the lists.

4.2.3 Menu Item Utility

The functions (**add-event-item** *function init_function label*) and (**add-history-item** *function init_function label*) install new items in the pop-up menus. The argument *function* specifies a function to call when the menu item is selected. It takes three arguments: the current window, the selected object (event or history), and the current state of the menu item (nil for off and t for on). *Function* returns the new state of the menu item. The argument *init_function* specifies a function to initialize the item when the menu activated. Its arguments are the current window and the selected object (event or history). The argument *label* specifies a label for the item. It may be any object that the (string) function will take. The functions (**remove-event-item** *label*) and (**remove-history-item** *label*) remove menu items.

4.2.4 Lisp Filter Utility

The display status of events and cross lines can be affected by the functions (**set-event-status** *window event status*) and (**set-xline-status** *window xline status*), where status can be one of *display*, *nodisplay*, or *highlight*. If no other filtering has been requested from the user interface this will be the mode in which the object will be displayed. If other filters have been requested the mode is computed by combining the requests as described in section 2.1.3.

4.2.5 Process Ordering Utility

The function (**set-proc-order** *window procorder*) specifies the set of processes to be displayed. The list *procorder* specifies both the set and the order in which it is to be displayed. A process can appear at most once.

4.2.6 Other Variables

The following global variables are also used by the utilities. Changing them can be dangerous.

win-hist-table This is an association list binding histories to windows.

win-table* and *hist-table These are lists of the association lists managed by the association utility. The next available id_number is kept in *win-assoc-id* and *hist-assoc-id*. The initialization functions are kept in *init-win-assoc* and *init-hist-assoc*.

event-function-table* and *hist-function-table These are tables that hold the functions bound to dynamically created menu items in the event and history menus.

procorder This is the list of process numbers that specifies the set of displayed processes.

4.3 History and Window Functions

These functions are used to control multiple histories and windows.

```
macro: (hist-draw-display hist)              /* Redraw the graph */
macro: (init-history filename &optional defaults) -> history /* Load a history */
macro: (new-window &optional displayname) -> window          /* Open a window */
macro: (moviola-bind window history)         /* Bind a history to a window */
macro: (free-hist history)                   /* Free a history */
macro: (free-win window)                     /* Free a window */
```

5 Instrumented Synchronization Packages

Moviola is structured to support the simultaneous use of multiple instrumentation packages. Each instrumentation package requires that a backend be written for it and as many backends as are needed can be compiled into *Moviola* .

There are currently two instrumented synchronization packages in use. We will describe the package we use for programs on BBN Butterfly (TM) Parallel Processor that run directly with the Chrysalis (TM) operating system. There is also a package for the Lynx [Scott, 1986] programming language on top of that base.

In the Chrysalis backend there are three types of shared object. The first type is a shared memory object whose structure is defined by the user. The package provides primitives for a single writer, multiple readers locking protocol. The second type is a Chrysalis shared event object. A Chrysalis

event can be thought of as a mailbox owned by a single process. The event can hold a single long integer at one time. Any process can post to any event, but only the owner can read from one. The third type of object is a Chrysalis shared `dualq`. This is a shared object that holds a head and tail pointer to a queue. A process can atomically write to either the head and the tail of a `dualq`, but a process can only read from the head of the queue. Hence the `dualq` can double as a stack. The following event types are generated by the package.

5.1 Process Header Events

MASTER_PROCESS: operation type = 0
objectID = a process ID

In each execution the first process to start is the master. The master process header contains the time the process took from start to finish, the number of events in the process, and a pointer to the data structure representing the synchronization history. This event is not actually read from a data file, but is created by *Moviola* to mark the beginning of the process.

PROCESS_HEAD: operation type = 1
objectID = a process ID

All other processes are marked with process-head events.

5.2 Shared Memory Objects

The package provides a set of operations for a single-reader, multiple-writers locking protocol to be used with shared memory objects. Some of the operations for obtaining locks check that a user-defined predicate is satisfied before actually obtaining the lock and returning to user code. These primitives are implemented by evaluating the predicate in a polling loop. The process holds an exclusive lock on the object while the predicate is being evaluated. If the operation allows concurrent access, the lock is then weakened before the primitive returns. The purpose of these primitives is to provide direct support for events consistent with complex communication primitives. For example, to write into a message buffer, a process must wait until it can get a write-lock and until the previous message has been removed. By including polling within the primitive operation the interval from the first attempt to obtain the lock to success appears as a single event in the history.

MEMORY_POLL_WRITE_START: operation type = 2:
objectID = a shared memory object ID

A memory-poll-write-start event returns with a write lock.

MEMORY_POLL_READ_START: operation type = 3:
objectID = a shared memory object ID

A memory-poll-read-start event returns with a read lock.

MEMORY_POLL_NULL: operation type = 4:
objectID = a shared memory object ID

A memory-poll-null event returns when the predicate is satisfied but does not obtain any locks.

MEMORY_DELETE: operation type = 5:
objectID = a shared memory object ID

A memory-delete event deletes the shared object.

MEMORY_READ_START: operation type = 17:
objectID = a shared memory object ID

A memory-read-start event is a blocking operation that obtains a read lock for the shared object.

MEMORY_READ_END: operation type = 18:
objectID = a shared memory object ID

A memory-read-end event releases the read lock on the specified object.

MEMORY_WRITE_START: operation type = 19:
objectID = a shared memory object ID

A memory-write-start event is a blocking operation that obtains a write lock on the object.

MEMORY_WRITE_END: operation type = 20:
objectID = a shared memory object ID

A memory-write-end event will release a write lock.

5.3 Events on Chrysalis Events

EVENT_RESET: operation type = 6:
objectID = a Chrysalis shared event object ID

An event-reset event clears and empties a specified mailbox.

EVENT_POST: operation type = 7:
objectID = a Chrysalis shared event object ID

An event-post event places the letter in the specified mailbox if it is empty.

EVENT_DATA: operation type = 8:
objectID = a Chrysalis shared event object ID

An event-data event returns the letter from a specified mailbox. If there is no letter, the reset value is returned.

EVENT_DELETE: operation type = 9:
objectID = a Chrysalis shared event object ID

An event-delete event deletes the mailbox.

EVENT_WAIT: operation type = 10:
objectID = a Chrysalis shared event object ID

An event-wait event blocks until a letter is received in one of its mailboxes. It then returns the ID of the mailbox in which it was received.

EVENT_MWAIT: operation type = 11:
objectID = a Chrysalis shared event object ID

An event-mwait event will block until a letter is received in one of several specified mailboxes. It then returns the ID of the mailbox in which it was received.

5.4 Events on DualQueues

DUALQ_ENQ: operation type = 12:
objectID = a Chrysalis shared dualq object ID

A dualq-enq event places a value on the dualq at the head or tail as specified. A bus error occurs if the dualq was full.

DUALQ_TRY_ENQ: operation type = 13:
objectID = a Chrysalis shared dualq object ID

A dualq-try-enq event does the same as a dualq-enq except it returns FALSE if the dualq was full.

DUALQ_WAIT: operation type = 14:
objectID = a Chrysalis shared dualq object ID

A dualq-wait event blocks until it can return a value from the head of the dualq.

DUALQ_POLL: operation type = 15:
objectID = a Chrysalis shared dualq object ID

A dualq-poll event returns a value from the head of the dualq if one exists.

DUALQ_DELETE: operation type = 16:
objectID = a Chrysalis shared dualq object ID

A dualq-delete event deletes the dualq.

5.5 Other Events

USER_DEFINED_TAG: operation type = 21:
objectID = no object ID

A user-defined-tag event can be inserted by the user at "interesting" points in the execution of a process.

SYSTEM_DEFINED_TAG: operation type = 22:
objectID = no object ID

A system_defined_tag event is generated by the system at "interesting" points in the execution of a process.

EVENT_ERROR: operation type = 23:
objectID = no object ID

An event-error event occurs when the process is killed during the execution of an event.

DIVISION: operation type = 24:
objectID = no object ID

A division event is a synthetic event inserted by *Moviola* in every process when no process is doing anything interesting (not touching any shared object) for a reasonable amount of time. These make the display more compact.

A Directories and Files

On the shared file system in the Computer Science Department at the University of Rochester the following directories and files are of interest:

/u/replay is the repository for PPUTTS-related material.

/u/replay/news contains documentation for recent updates of PPUTTS. This includes both the text for appendices updating this guide as well as notes on recent changes that have not yet been put in final form.

/u/replay/lib contains the standard versions of the *Moviola* initialization files. It also contains the files `pputts.lsp` and `.tools` used in the initialization of the PPUTTs version of Lisp. The directory containing PUTTs manual pages is also located here.

/u/replay/bin contains symbolic links to the executibles.

/u/replay/src/moviola contains all of the *Moviola* source code and executibles.

moviola is the standalone version of *Moviola* . There is a link from /u/replay/bin.

moviola.o is the *Moviola* tool loaded by Lisp.

cast1.c and include/cast1.h contain the source code for the Chrysalis backend.

c-int.c, lisp-int.lsp, c.lsp, *.lsp define the interface between Lisp and and tools written in C, specifically *Moviola* .

/u/replay/src/toolkit contains the locally modified version Kyoto Common Lisp that forms the basis for the Toolkit.

/u/replay/src/lisptools contains the tools written in Lisp.

References

[Bernstein *et al.*, 1987] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

[Duda *et al.*, 1987] A. Duda, G. Harrus, Y. Haddad, and G. Bernard, "Estimating Global Time in Distributed Systems," *Proceedings of the 7th International Conference on Distributed Systems*, pages 299–306, September 1987.

[Fowler *et al.*, 1988] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey, "An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multi-Processors," *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 163–173, May 1988.

[Gettys and Scheifler, 1986] Jim Gettys and Robert W. Scheifler, "The X Window System," *ACM Transactions on Graphics*, V5(2):79–109, April 1986.

[Lamport, 1978] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, V2.1(7):558–565, July 1978.

[LeBlanc and Mellor-Crummey, 1987] Thomas J. LeBlanc and John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[Scott, 1986] M.L. Scott, "LYNX Reference Manual," Technical Report BPR 7, Computer Science Department, University of Rochester, Rochester, NY, August 1986, (revised).

[Yuasa and Hagiya] Taiichi Yuasa and Masami Hagiya, *Kyoto Common Lisp Report*.